Case-based Tutoring in Virtual Education Environments

Patrick M Regan
North Dakota State University
413 E 82nd Street
New York, NY 10028
patrick_m_regan@hotmail.com

Brian M. Slator
North Dakota State University
IACC Building #258
Fargo, ND 58105
slator@cs.ndsu.edu

ABSTRACT
Virtual education environments are gaining popularity as tools to enhance student learning. These environments are often used to allow students to experience situations that would be difficult, costly, or impossible in the physical world. North Dakota State University (NDSU) provides students with environments to enhance their understanding of geology (Planet Oit), cellular biology (Virtual Cell), retailing (DollarBay), and history (Blackwood). In order to maximize the learning potential of each individual student, an ideal environment needs to provide customized lessons to that student based on his or her individual performance. One method to address this requirement is the use of case-based reasoning which is used to monitor student performance, track progress throughout an environment, compare the student to other students in the same environment, and create customized tutor dialogs to communicate this information to the student in the form of individual tutor lessons. An example of case-based lesson building software that meets the above requirements can be observed in the current DollarBay retailing environment at NDSU.

Categories and Subject Descriptors
J. Computer Applications: J.4 Social and Behavioral Sciences

General Terms
Algorithms, Design, Economics, Experimentation, Human Factors, Theory

Keywords
Case-based Tutoring, Computers in Education, Intelligent Systems, Multimedia Applications

1. INTRODUCTION
The traditional teaching environment is usually thought to be that of a classroom: a single teacher giving lectures to a group of students who are expected to use their notes and textbook to prepare for periodic examinations and demonstrate that they have learned. Technology provides an alternative to this scenario. One of the ways technology can be used to supplement learning is through the construction of virtual education environments to simulate scenarios that may be difficult for students to experience in the physical world. Using the Internet, students can access these worlds remotely, be it in a classroom or in the solitude of their own dwelling. At North Dakota State University (NDSU), the World Wide Web Instructional Committee (WWWIC) is engaged in research aimed at developing virtual education environments to assist in the education and growth of students [7]. Some of the key factors that lead to success of these environments at NDSU are the theory of role-based environments on which they are based, the use of graduate and undergraduate students in the development process, the use of the environments in actual classes, and the application of knowledge from one environment to the others. One of the major goals of WWWIC research is to find ways to provide tutoring agents to communicate "expert stories" to students as they progress through the environment. These agents should monitor the student and send advice on an "as needed" basis while being careful to never insist upon or block any course of action [8].

Although several of the virtual education environments at NDSU provide some basic tutoring functionality to students, none had implemented a completely functional case-based system except DollarBay. This system provides complete analysis of student behavior based on selected attributes and a message delivery mechanism. This paper describes the design and functionality of the case-based tutoring system implemented in DollarBay. This system provides the means to generate personalized lessons for each student participating in the DollarBay environment. In addition, it provides a framework that may be used to implement similar functionality in the other virtual education environments at NDSU with a minimum of coding.

The DollarBay simulation is based upon a client/server paradigm. The server side of DollarBay consists of a server program and a database. It is permanently connected to the Internet and allows other users to connect at any time, from any location, to the DollarBay environment. The database for DollarBay is implemented in LambdaMOO [1]. The database contains representations of all of the objects in the DollarBay environment, including the MOO programs necessary to give the objects their specific behavior. On the client side, the student interacts in the DollarBay environment by using a graphical user interface (GUI). The GUI window is generated by a Java applet and serves as the connection to the LambdaMOO server. This allows the student to interact with the environment by pointing, clicking, and selecting objects as well as typing text.

1.1 Playing the DollarBay Game
As players engage themselves in the DollarBay game, they are assigned a location and must decide what to sell, what level of service to offer, how much to spend on advertising, how much to

CVE '02, September 30-October 2, 2002, Bonn, Germany.

stock, who to buy from, and what price to set to appear attractive to the customer agents [7]. In order to simulate an economic environment, time is divided into "virtual weeks". Each week the customer agents are given a shopping list representing a weeks worth of demand for various products. After each week has concluded and the shopping lists are exhausted, each agent assigns new attractiveness ratings to each store based upon the past week's experience and new shopping lists are created for the upcoming week [7].

At the end of each virtual week the weekly calculation charges players for their weekly expenses, recalculates the customer agent motivations as described above, and updates each of the player cases. At the end of a player's life, they are retired to the Hall of Fame. The Hall of Fame is a place where players are moved when they graduate from the game by reaching a profit goal or are inactive for a long period of time. Players are moved to the Hall of Fame by the Reaper, who is sent out periodically to retire graduated and inactive players. The Reaper is responsible for recycling all of the objects related to the player, such as store, company, ads, and products. Recycling makes object numbers available for reuse at a future date as new objects are created. The Reaper also moves the player's active case to historical cases for future reference by case-based tutors.

2. BACKGROUND

2.1 Overview of LambdaMOO

MOO is an abbreviation for a Multi User Dungeon Object Oriented or Multiuse Object Oriented system. It is designed to be a network accessible, programmable, interactive, multi-user system that is well suited to the construction of text-based collaborative software most commonly used as multi-participant low bandwidth virtual reality [1]. It contains a small, simple language that is designed to be easy to learn and supports expressions, looping, control structures, and built-in functions. LambdaMOO models virtual reality by representing virtual world entities (tutors, cases, stores, players, etc) as objects. It supports other value types as well (integers, strings, etc) but the objects are the backbone of a LambdaMOO database [1]. A property is used to store an arbitrary MOO value. Verbs are LambdaMOO programs associated with a particular object.

In addition to the functionality described above, LambdaMOO also includes a built in command parser. Any time the command parser is invoked, the first word is always taken to be a verb. After identifying the form of the command, the parser may then proceed to execute it using one of the built-in commands, or by looking up the MOO objects representing the direct and indirect objects and then executing the command [1]. LambdaMOO is also an object-oriented language.

2.2 Case-based Reasoning

"Case-based reasoning means reasoning based on previous cases or experiences" [2]. The purpose of this approach is to form a judgment about a situation based on cases that are already classified. In order to perform case-based tutoring, one must understand the definition of a case. A case is an example of a past problem and its solution. In DollarBay, cases are sets of both abstract and concrete attributes that represent the status of a player's business strategy. Case-based tutoring is intended to enhance the effectiveness of student learning via educational simulation environments.

The design of an educational simulation environment is based upon several core principles, the first being that the environment is role based. This allows the student to fully assume the role of a character trying to accomplish some predetermined goal. The second principle is that the environment itself must be immersive. As the student assumes a particular role in the environment, he or she becomes immersed in it and starts to think and make decisions like that character would. The simulation must also be highly interactive. Environments must have a goal or objective to accomplish. Finally, it must be game like [6].

2.3 Intelligent Tutoring

The overall goal of intelligent tutoring is to focus on developing and employing intelligent agents within multi-user distributed simulations to help provide effective learning experiences [8]. Examples of diagnostic tutoring may be seen in Planet Oit [7]. For example, the science tutor looks at the decision making process that a player follows while trying to properly identify a material, and what experiments were performed on the material, the tutor is able to identify students who have made "lucky guesses" and let them know that they did not follow the proper process in getting to their answer.

Rule-based tutoring at NDSU was at one time functional in DollarBay. A rule-based tutor functions by knowing a set of rules about a domain, monitoring student action for any indication of breaking one of the rules, and then visiting the student to present a warning [6]. For example, one of the rules that the now defunct rule-based tutor in DollarBay monitored was if a student had set their prices to an excessive markup. In such an instance, the tutor would send a message to the student informing them that they may be setting their prices too high [6].

An example of case-based tutoring outside the NDSU realm is the Georgia Tech Case-Based Intelligent Tutoring System (GT-CBITS). This system is used to demonstrate the importance of critical information to airplane pilots by using stories of difficult situations or incidents encountered by other pilots. The topics presented can range from problems arising from the complex nature of aircraft, the dynamic nature of the aviation environment, or new/changing features of an aircraft [3]. Case-based tutoring at NDSU has presently only been implemented in the DollarBay virtual world.

2.4 Case-based Tutoring in DollarBay

The DollarBay case builder is centered on a set of verbs that are all defined on an object named $g.gencase. These verbs control the creation, initialization, comparison, and updating of all cases. The children of the $g.gencase object are the cases themselves. Case updates are carried out periodically during the "weekend calculations" in the game as well as prior to any tutor message generation. In addition to the $g.gencase object, a $g.tutor_agent object exists for the purpose of generating tutor messages. This object owns the verbs that handle the construction of tutor messages and the scheduling of tutor visits. Tutor visits are scheduled during the execution of a player's $g.retail_player:confunc verb, which runs each time the player connects to the game[4].

3. ARCHITECTURE OF CASE-BASED LESSON BUILDING SYSTEM

The backbone of any case based tutor system is an effective case matching procedure coupled with a substantial case library. The fundamental building block of any case-based system is a case. All cases in DollarBay divided into four distinct groups:

1. Historical Cases ($g.historical_cases) – a copy of the final case for each store when the player is reaped and moved to the Hall of Fame.
2. Case Library ($g.case_library) – a copy of each unique case that has occurred in the history of DollarBay. Each of these cases represents a unique combination of the eight abstract properties of a case
3. Prototypical Cases ($g.protocase) – the five prototypical cases that are used by the tutor to determine abstract case matches. The descriptions of these prototypes currently are:
    a. Overcautious – player has plenty of money and is not using it to stock rapidly or expand product lines
    b. Overspender – player is too liberal with money (expensive employee, expensive ads, loans) and has no clear price advantage or plan
    c. Best Player – player who is profitable and not doing anything obviously wrong
    d. Barking up the Wrong Tree – player products and services do not match ads and the prices are not appealing to target group
    e. Foolish Squanderer – player is bleeding money and has no clue about how to run a business

In DollarBay, each case shares a set of similar abstract attributes that define it. A list of the similarity measures, their acceptable values, and a brief description of how each is calculated follow:

1. product_focus {"not evaluated", "tight", "moderate", "wide"} – looks at a list of the current product families and number of products stocked for each store. Computes a "focus factor" based on ratio of number of types to total quantity stocked
2. restock_plan {"not evaluated", "understocking", "acceptable", "overstocking"} – estimates annual sales based on accounting records and annual revenue from the price of items currently on order. Creates a ratio of projected revenue to projected sales
3. ad_cost {"not evaluated", "low", "affordable", "high"} – uses the liquid assets and advertising costs of a company to compute the average advertising cost per day. Determines the percent of available cash used to purchase advertising
4. ad_target {"not evaluated", "fuzzy", "okay", "focused"} – gets a set of cluster groups based on what ads are running, looks at the sales records, and determines the effectiveness of each ad based on the attractiveness of the ad to each cluster group from the sales
5. staff_level {"not evaluated", "underskilled", "appropriate", "overskilled"} – generates a ratio of current staff cost to total sales
6. cash_reserves {"not evaluated", "low", "moderate", "high"} – gets the current cash on hand, the total of all expenses incurred, and calculates the average daily expenses
7. price_margin {"not evaluated", "low", "moderate", "high"} – obtains the current cost and price for each product, computes the markup for each
8. liability {"not evaluated", "good", "risky", "very risky"} – gets the daily loan payments and generates a ratio of payments to income

In addition to the abstract similarity attributes of a case listed above, each case contains additional attributes that are used to assist in case-based lesson building. An explanation for each follows:

1. active_similarity – contains a ranked list of active player cases sorted by the sum of their abstract similarity values. This attribute is used to determine the most similar active case to the current case
2. prototype_similarity – contains a ranked list of prototypical cases sorted by the sum of their abstract similarity values. This attribute is used to determine the most similar prototypical case to a given case
3. product_list – contains a list of products that are currently in stock for an active player store
4. prototype_family – contains a pair of elements stored in a list. The first element is the object number of the prototypical case that a case matches the most. The second element is a descriptor of how closely the case matches the prototypical case contained in the first element
5. active_tutor_dialog – used during similarity calculations to store a list of information that is later used to generate a tutor message
6. last_tutor_visit – displays a list containing the turn and timestamp of the last tutor visit a player received
7. library_match_history – a list of lists. Each sub list contains a game turn and library case object number for the first time the current case matched the given library case. This attribute is used to track which library cases an active case has matched over time
8. prototype_tutor_dialog - used during similarity calculations to store a list of information that is later used to generate a tutor message

3.1 Prior Work

At the outset of the implementation of this project, a portion of the work had already been completed. The current case-based message builder for DollarBay is based upon the original work.

3.2 Weekly Case Update

At the end of each virtual week in DollarBay, a set of weekly calculations is run. As a part of this process the weekly case update is performed, which brings the active cases for all of the valid player stores up to date by calculating the current values of the eight similarity measures described above. If an active case for each player store cannot be retrieved, a new case is created for that store. The case library is also updated with each active case.

3.3 Update Case Library

During a case library update the player's case is first compared to every case in the library. During the comparison, the eight abstract similarity values of the player case are compared to the respective values of each library case. If an exact match is

identified, the player's case is added to the match_history property of the library case. The property contains a list of the first time any player case has matched a specific library case. If the player's case has already been matched to the current library case at some point in the past, no new entry is made to the match_history of the case. If the player's case has never been recorded in the match_history of the matching library case, a new entry is generated using the generate_history_entry algorithm. The entry generated by this algorithm contains the current turn, the store owner object number, the store owner name, and the product families that are currently stocked in the store.

If, after comparing the player's case to the entire case library, no exact abstract similarity value match is found, a new library case is created. The eight abstract similarity properties for the new library case are then set to the same values as the player case and the first entry, which contains the player's information, is made to the match_history list for the new library case.

The final step in updating the case library is to update the library match history for the player case. This property contains a list that keeps track of every library case the player case has ever matched. It is updated by first looking to see if the current entry will be the first. If so, a new entry, which consists of the turn number and the object number of the matching library case, is created. If the entry will not be the first, the last entry in the list is examined to see if its library case is the same as the library case that currently matches the player case. If the last entry does contain the same library case as the current match, no new entry is created. If the last match is not the same as the current library case match, a new entry is generated and added to the end of the list.

3.4 Determining Case Similarity

In order to construct a meaningful tutor message, cases must be compared. This comparison is completed in the hopes of determining similar cases from which information is drawn to construct a message. To accomplish this, the tutor messages depend upon comparisons of the player case to the prototypical cases and the other active cases. The compute_similarity_value verb is a general-purpose comparison engine used to determine the similarity of two cases based upon their eight abstract similarity properties. The similarity is constructed by comparing each of the abstract properties for the two cases. A direct match results in 10 points, thus creating in a maximum possible similarity value of 80 for two identical cases [4].

The algorithm begins by looking at each abstract attribute on the two cases. If either case is "not evaluated" for the attribute currently being examined, the attribute is skipped and the next attribute examined. As the comparison is carried out, lists containing matched attributes and unmatched attributes are constructed as is the total similarity point value. The algorithm is able to handle comparison of prototypical cases with multiple values for an attribute by keeping track of the closest comparison between the other case's value and the multiple attribute values of the prototypical case in such an instance. The verb then checks to see if the comparison involves prototypical cases. If it does, the prototype_tutor_dialog of the active case being compared is set to a list containing the prototypical case object number, a list of the matched attributes, and a list of the unmatched attributes. This data is important for future tutor dialog construction. The verb completes by returning the similarity value that is the result of the comparison.

3.4.1 Prototypical Case Similarity

An active case's prototypical similarity is stored as a list in its prototype_similarity property. In addition, each active case contains a prototype_family property that describes how strongly it matches the prototypical case that it is most similar to. In order to set the values of these properties, the eval_prototype_ similarity_and_sort verb is called on an active case.

The first step in determining an active case's prototypical similarity is to clear the tutor_dialog property of the case. This property is used to store information generated during the similarity check for later use in constructing a tutor message. The active case is then compared to each prototypical case by calling the compute_ similarity_value verb (explained in the previous section).

As the active case is compared to each of the prototypical cases, the result of each comparison is stored in a similarity list as a list containing the object number of the prototypical case and the similarity value. Upon completion of the comparisons, this list is sorted by the similarity values from highest to lowest, and the resulting list is saved as the prototype_similarity property of the active case. The final step in determining prototypical similarity is to set the prototype_family of the active case.

The algorithm that determines the prototype_family of the active case begins by generating a list of the highest ranked prototypical cases. This list is called the family list and is generated in case there is more than one prototypical case with the highest similarity in the active case's prototype_similarity list. In the event that there is a single entry in the family list, the prototype_family of the active case is set to the prototypical case of that entry. If there is more than one prototypical case in the family list, the algorithm compares the entries to determine which should be set as the prototype_family for the active case.

The final step in setting the prototype_family of an active case occurs by determining the strength of the match between the active case and the prototypical case. If the similarity value for the matched prototype family is high (in the top 25% of the similarity value range), the match is classified as "strong". If the similarity value is between 50% and 75% of the maximum value, the match is classified as "weak". Any prototype_family based upon a similarity value of less than 50% of the maximum, of which there should be very few, is classified as "very weak". After the strength of the match is determined, it is stored in the active case's prototype_family property along with the object number of the most similar prototypical case.

3.4.2 Active Case Similarity

An active case's similarity to the other active cases is stored as a list in its active_similarity property. The value of this property is set by the eval_active_similarity_and_sort verb.

The first thing the eval_active_similarity_and_sort algorithm does is compares the player's active case against every other active case by calling the compute_similarity_value verb. The results of each similarity computation are stored as a list of ordered pairs containing the object number of the compared active case and the similarity value. Upon completion of the comparisons, this list is

sorted by the similarity values from highest to lowest, and the result is saved as the active_similarity property of the active case. Each entry in the list contains the object number of the active case that was compared, the similarity of the case to the player's case, and two zeros that are used as placeholders for the concrete similarity results.

3.4.3 Concrete Active Case Similarity

In addition to the active similarity value based on the eight abstract similarity properties it is useful to understand how a player compares to other players in terms that are more concrete. The intent of the eval_concrete_active_similarity verb is to examine active players of the highest abstract correlation in these concrete terms and provide additional comparison criteria to further refine the measure of similarity between them.

The algorithm begins by first determining the similarity value of the highest ranked active case in the player's active_similarity list (the first case in the list, since it has been sorted). It then proceeds to compute a concrete similarity value for every entry in the player's active_similarity list that has a similarity value equal to the first entry. The concrete similarity value is returned from a call to the compute_concrete_similarity_value verb. It should be noted that with ten or more players in the game there are often multiple active cases with the same active similarity in a player's active_similarity list, and thus the need for this concrete comparison.

The compute_concrete_similarity_value verb determines a concrete similarity value by comparing several concrete attributes of the player's case to the active case being compared. It keeps a running total of similarity points based on the results of each concrete comparison and returns it upon completion. The concrete similarity measures are:

1. number_of_product_families_stocked - examines the product list of each case and generates a count of product families for each by examining the ancestors of each product in the product lists.
2. product_families - examines the product list of each case and generates a list of product families for each
3. number_of_products_stocked - computes the length of the product list for each case
4. average_markup - looks at all of the owned_objects of a store's owner, determines if each is a shipment, calculates the percent markup of the shipment, adds the markup to a running total, and increments a counter. When the verb is finished looking at all of the owned_objects, it then divides the total markup by the counter to get the average markup percentage for products stocked
5. cash - looks at the cash property of the company for each case
6. player_age - looks at the first_connect_time property for the owner of each player case
7. items_sold_last_week - if an employee is found for both cases, the comparison begins by examining the lw_sold property on each employee. A total of items sold is then computed for each case using this property
8. sales_lost_for_not_stocked - if both cases have valid employees, it examines the lw_not_stocked property of each. A running total of lost sales for each employee is then computed
9. employee_type – looks at the parent of the employee for each case to determine the employee type
10. ad_type – looks at the types of ads run by the owner of each case
11. net – looks at the net_worth property of each case's company

There is a verb to handle each of the concrete comparisons and each returns a similarity score for that comparison based upon the strength of the match. The score returned by each verb is currently three points for a strong match, one point for a weak match, and no points for an insignificant match. Anytime a strong match is identified, a string describing the situation is appended to the active_tutor_dialog property of the player's case for later use in tutor message generation.

After the compute_concrete_similarity_value verb returns a concrete similarity value, the eval_concrete_similarity verb finishes by updating the active_similarity property of the case with new data. It was previously explained that the eval_active_similarity_and_sort verb placed two zeros as placeholders in each entry of a player's active_similarity property of their case. The first of these placeholders is now filled with the concrete similarity score for the case, and the second with the sum of the active similarity and the new concrete active similarity scores.

4. TUTORING

This tutoring process is currently triggered whenever a player logs in to DollarBay. During execution of the confunc (a function called at connect time), several criteria are checked to see if the player should receive a tutor. The player must meet all of the criteria to receive a tutor visit. The criteria are:

- Is the player a valid player?
- Is the tutor_enable property of the player set?
- Is it true that the player has received no tutor visits yet this week?
- Has the player been playing for more than twelve hours?

If all of the criteria are met, the process begins by calling the schedule_tutor_visits verb. This verb coordinates the scheduling and sending of tutors to players that have just connected to DollarBay and limits the number of tutors that are sent at one time in order to avoid overloading the server. It begins by adding the connecting player to the list contained in the needs_tutor property of the tutor agent. It then forks a process to handle the sending of the tutor to the player(s) in that list. The process looks to see if players are in the needs_tutor list (at a minimum, the player that just connected should be in the list). It will continue processing all players in this list until it is empty. The forked process then looks to see how many tutor processes are currently running. If less than four tutors are currently active the process continues, otherwise it returns and lets the four active tutor processes handle the player who has just connected [4].

If there are less than three tutors running, the new process begins by taking a snapshot of the players in the needs_tutor list. The process then finds the player in the list with the lowest net worth, operating on the theory that the player who is doing the worst needs tutoring the most. That player is then removed from the list and the send_tutor verb is called to dispatch a tutor to that player. Upon returning from the send_tutor verb, the number_of_tutor_visits_this_week property of the player is

updated to document that the player has been visited by a tutor this week and the needs_tutor list is examined to see if any players needing tutoring remain in the list. If so, the process repeats itself. Otherwise, the forked process terminates.

The first thing the send_tutor verb does is increment the active_tutors property of the tutor_agent. It then proceeds to call update_active_case, eval_active_similarity_and_sort, and eval_concrete_active_similarity (each described above) to completely update the player's case. The player is then examined to see if they are a perfect match to the best player prototypical case. If it is found that they are indeed in the "best player" class, the active_tutors counter is decremented and the verb returns without doing anything.

If the player is not found to be a best player, a message is constructed for transmission to the player. The message begins with an introduction from the tutor. Following the introduction from the tutor, the build_active_message, build_prototype_message, and build_historical_message verbs are called to generate the personalized lessons for the player. These customized lessons are appended to the greeting and sent to the player as a tutor message via the show_note verb. The show_note verb pops up a new window containing the tutor message on the player's screen. The player may choose to read or ignore the message, and may keep the window open for reference throughout their playing session. The process completes by updating the last_tutor_visit property of the player's case with the turn and time that the tutor message was sent. It also decrements the active_tutors property before it completes the process of sending the tutor to the player.

4.1 Building Messages

The build_active_message verb creates a customized tutor lesson for the player based upon the store that is currently most similar to the player's store. It begins by looking at the active_similarity list on the player case to find the store that is most similar to the player. A list of similar traits is then grabbed from the player's active_tutor_dialog, which was generated during the case update. The verb then constructs a message of the following nature:

> *"If I were to pick a store that you are most similar to, it would be **XXX'S** which is located in **XXX'S_NEIGHBORHOOD**. The things you have most in common with **XXX'S** are that you both:*
>
> ***- SOME SPECIFIC TRAIT MATCH INFO***
>
> ***- MORE SPECIFIC TRAIT MATCH INFO***
>
> ***- ETC.***
>
> *Keep in mind that this may be good or bad, depending on how **XXX'S** is performing."*

The items shown in bold are replaced with information specific to the matched player store, and the lesson is returned to the send_tutor verb where it is appended to the tutor message.

The build_prototype_message verb creates a custom lesson for the player based upon the prototypical case that they are most similar to. The lesson is pieced together to look something like this:

> *"You **STRONGLY OR WEAKLY** remind me of a player who was **TYPE**. In general this means that:*
>
> ***- GENERAL_TYPE_MESSAGE***
>
> *My advice for you is to:*
>
> ***- SPECIFIC_MESSAGE***
>
> ***- ANOTHER_SPECIFIC_MESSAGE***
>
> ***- ETC."***

The types and corresponding general messages are:

1. overcautious - you are not using your cash to expand or keep your stock level where it should be
2. overspending - you are far too liberal with your money
3. one of the best players - you are doing a great job
4. barking up the wrong tree - your products and service do not match your advertising, and your price is not appealing to your target group
5. a foolish squanderer - you are breaking the bank and are not using ANY sound business practices

Each of these messages is intended to give the player a general sense of their behavior pattern. The message for each player is based upon which prototype family they belong to.

The specific messages added to the end of the lesson are described in Table 1 below. The player will receive eight of these messages (one for each abstract similarity measure). Each message provides a detailed explanation on how a player should alter their behavior to better succeed or an affirmation of good behavior for each of the abstract similarity properties. The addition of these specific messages completes the construction of the prototypical tutor lesson, which is then returned to the send_tutor verb.

Table 1. Specific Prototypical Case Tutor Messages

Attribute	Value	Message
product_ focus	tight	- stay focused on the types of products you are selling... good job!
	moder-ate	- don't change to any new types right now
	wide	- stick with selling the same types of products... don't change between different types
restock_ plan	under-stocking	- stock more quantity of the products you sell
	accept-able	- keep stocking the same quantity products you sell... good job!
	over-stocking	- stock less quantity of the products you sell
ad_ cost	low	- spend more of your income on ads
	afford-able	- keep spending the same amount of your income on ads... good job!
	high	- spend less of your income on ads
ad_ target	fuzzy	- use market research to properly target your ads for the consumers you need to reach
	okay	- use market research to improve the targeting of your ads and reach the right consumer groups
	focused	- keep targeting your ads to reach the right consumer groups... good job!

staff_ level	under-skilled	- hire a more skilled employee to meet the needs of your customers
	appro-priate	- keep your current employee... good job!
	over-skilled	- hire an employee with less skill and stop wasting your money on the one you have
cash_ reserves	low	- try to keep more cash on hand
	moder-ate	- keep the same amount of cash on hand... good job!
	high	- try to spend more of your cash to improve your situation
price_ margin	low	- keep your prices competitive... good job!
	moder-ate	- improve your prices as they are only marginally competitive
	high	- lower your prices to get competitive
liability	good	- keep your liability at an acceptable level... good job!
	risky	- consider reducing your liability
	very risky	- reduce your liability as soon as you can

The final lesson to be appended to the tutor message is based upon the history of the player. This lesson is constructed by going through the player's library_match_history and examining all of the prototype case families a player has matched over time. The families are chronologically ordered, the duration of time spent at each is calculated, and a lesson of the following format is constructed:

> *"Going back in time from present to distant past, it seems that you have been:*
>
> *- **TYPE STRONGLY_OR_WEAKLY** for **X_NUMBER_ OF** weeks*
>
> *- **TYPE STRONGLY_OR_WEAKLY** for **X_NUMBER_ OF** weeks*
>
> *- **ETC.**"*

The type correlates to one of those described above. If the player is currently "one of the best players", a message stating, "You are doing a great job! Keep it up" is appended to the end of the lesson. If the player was previously "one of the best players", but has slipped into another prototypical family type, a message stating, "*You were doing a great job! What happened???*" is appended to the end of the lesson.

5. EVALUATION

A controlled study of the case-based tutor in Dollar Bay was conducted during June, 2002. A total of 16 stores were fabricated according the eight strategies described in the schema described in Table 2.

Table 2. Player Types in DollarBay

Bargain Focus big ticket (BF_bt): One product, lowest quality, lowest service, no ads	**Luxury Focus big ticket (LF_bt):** One product, highest quality, highest service, highest ad
Bargain Comprehensive big ticket (BC_bt): Many related products, lowest quality, lowest service, no ads	**Luxury Comprehensive big ticket (LC_bt):** Many related products, highest quality, highest service, highest ad
Bargain Focus low ticket (BF_lt): One product, lowest quality, lowest service, no ads	**Luxury Focus low ticket (LF_lt):** One product, highest quality, highest service, highest ad
Bargain Comprehensive low ticket (BC_lt): Many related products, lowest quality, lowest service, no ads	**Luxury Comprehensive low ticket (LC_lt):** Many related products, highest quality, highest service, highest ad

In this study the stores were created to be paired in neighborhoods according to their strategy. Therefore, each of the eight neighborhoods had two stores pursuing the same strategy. However, one of the stores stuck strictly to their "scripts" and received no tutoring, while the other store changed operating procedures by following the tutor's advice as closely as possible.

The stores were operated by a volunteer group of Governors School students. These students had over two weeks experience with playing the game, and thus were easily able to interpret both the scripts assigned to them and the advice the tutor provided. The students were given certain latitude in their operation of the stores. For example, if the tutor advised them to use market research to choose more expensive advertising, they were encouraged to decide for themselves what that meant, and what advertising to buy. In addition, they were free to choose their own employee within the bounds of their script, were advised to make their own decisions in terms of price setting (again within the bounds of their scripts and the tutors advice), and to decide for themselves about borrowing extra capital from the bank. It turned out that none of them chose to borrow.

After seven consecutive days (eleven virtual weeks) of this competition a snapshot of the simulation was taken.

Table 3. Final Player Scores in DollarBay

Store	Net Worth	Liquid assets	Profit	Liability
1. LC_lt_T's	$109,342	$107,902	$84,342	$0
2. BC_bt_T's	$84,472	$76,537	$59,472	$0
3. BF_bt_T's	$74,835	$74,835	$49,835	$0
4. BC_bt's	$72,665	$70,535	$47,665	$0
5. BF_bt's	$69,510	$66,485	$44,510	$0
6. BF_lt's	$68,815	$68,794	$43,815	$0
7. BC_lt's	$68,775	$65,825	$43,775	$0
8. BF_lt_T's	$68,674	$68,674	$43,674	$0
9. LF_bt's	$67,030	$48,580	$42,030	$0

10.BC_lt_T's	$66,675	$62,985	$41,675	$0
11.LF_bt_T's	$48,955	$48,955	$23,955	$0
12.LC_bt_T's	$35,925	$35,925	$10,925	$0
13. LF_lt's	$34,680	$34,680	$9,680	$0
14. LF_lt_T's	$33,290	$33,290	$8,290	$0
15. LC_bt's	$25,880	$25,440	$880	$0
16. LC_lt's	$19,530	$17,130	$-5,470	$0

The following observations are of interest:

1. The top three players are tutored, although the third is not significantly ahead of its counterpart.
2. The next two tutored stores are in the middle of the pack (8th and 10th place), and virtually tied with their counterparts (at 6^{th} and 7^{th}, and within 3% net worth in the worst case)
3. The next tutored store (11. LF_bt_T) is an anomaly in the data, and was removed from the analysis, as explained below.
4. The next tutored store is only in 12th place, but still three positions ahead and 28% more profitable than its counterpart LC_bt, in 15^{th} place.
5. The lowest tutored store, in 14th place, is virtually tied with it's counterpart in 13th place, with only 4% difference in net worth.
6. The lowest two players are non-Tutored.

Thus we see that case-based tutoring had positive effect, with no harm caused in any case. The tutor advice either proved to be highly beneficial, or showed no real difference in the course of this short study. That some tutored players still finished at the bottom of the scoreboard can be attributed to the weakness of the strategy employed by the player (as determined by the schema, above), and not the effect of tutoring itself. Indeed, the tutoring had significant effect in some cases. It is interesting to note that the first place and last place stores pursued the same strategy in the same neighborhood, and that the Tutored store proved to be 459% better ($89,812, or nearly 4 standard deviations).

5.1 Explanation of the Anomaly:

The Luxury Focus, big ticket pair (9. LF_bt and 11. LF_bt_T) were operating in head to head competition in Silver River. The tutored player finished in 11th place, only two positions behind its counterpart, but by a fairly wide margin, $18,075, which is a 27% difference in net worth. It turned out in this case that the untutored player attempted to buy advertising, according to the script, but the transaction failed. The player did not notice the failure,and played the entire week without advertising. The tutored player successfully purchased the advertising, as required by the script, which cost them a total of $38,400. This amount is far more than the difference in their net worth. Had the untutored player successfully followed the script, the difference could have conceivably been $20,325 in the other direction, in favor of the tutored player. This value cannot be flatly asserted, however, as the tutored player presumably received some benefit from their advertising, although clearly not enough to justify the cost. In any event, the two players were not on an equal footing for the purposes of comparison in this study, and they were removed from the data set.

6. ACKNOWLEDGMENTS

Our thanks to Acey Olson and James Walsh for providing a foundation to build our tutoring system upon. This portion of NDSU Worldwide Web Instructional Committee (WWWIC) research was supported by funding from the National Science Foundation under grant EIA-0086142

7. REFERENCES

[1] Curtis, Pavel. “LambdaMOO Programmer’s Manual for LambdaMOO Version 1.8.0p6”, Xerox, San Francisco, CA, March 1997.

[2] Kolodner, J. L. and Leake, D. B., “Case-based Reasoning: Experiences, Lessons, and Future Directions”, Chapter 2, AAAI Press/MIT Press, Menlo Park, CA.

[3] Palmer, E., NASA Ames, “Case-based Intelligent Tutoring for Pilots”, http://human-factors.arc.nasa.gov/projects/ihi/casebased.html.

[4] Regan, Patrick M. “Case-based Tutoring in Virtual Education Environments”, Master’s Thesis, Computer Science Department, North Dakota State University, Fargo, ND, 2002.

[5] Slator, Brian M., “Intelligent Tutors in Virtual Worlds”, 8th Annual Conference on Intelligent Systems (ICIS-99), Denver, CO, June 24-26, 1999, pp. 124-127.

[6] Slator, Brian M. and Golam Farooque, “The Agents in an Agent-based Economic Simulation Model”, Proc. 11th Int. Conf. on Computer Applications in Industry and Engineering (CAINE-98), Las Vegas, NV, November 11-13, 1998, pp. 175-179.

[7] Slator, Brian M., Paul Juell, Phillip E. McClean, Bernhardt Saini-Eidukat, Donald P. Schwert, Alan R. White, Curt Hill (1999). Virtual Environments for Education. Journal of Network and Computer Applications, 22(4), pp. 161-174.

[8] Zelenak, Jozef. “Conversation Construction for Agents in Educational Simulation Environments”, Master’s Thesis, Computer Science Department, North Dakota State University, Fargo, ND, 1999.